CS4411 Introduction to C

Owen Arden

owen@cs.cornell.edu

Upson 4126

Slide heritage: Alin Dobra → Niranjan Nagarajan → me

Why C?

- C is a great language for systems code
  - Low level operations for direct access to memory and control flow
  - High level abstractions for complex data structures and portable code
  - Direct control of system resources

But great power can corrupt…

Goals

- A “nudge” in the right direction
  - Learn by doing!
- Show a few correct examples and describe a few common mistakes
- Give you enough information so you can compile-test-debug on your own

hello.c

```c
/* Hello World program */
#include <stdio.h>

int main(void){
  printf("Hello World.\n");
  return 0;
}
```

Try it out

- Using your favorite editor, create hello.c
- From a VS2008 command prompt run:
  - cl hello.c
- Now run hello.exe

How to learn a new language

- Draw from experience
  - Many languages are similar
  - Learn a lot of languages!
- Anticipate generic language features
  - Control primitives (for, while, etc)
  - Data types (int, char, etc)
- Discover the strengths of the language
  - Don’t use a square peg for a round hole

Common Syntax with Java

- Operators:
  - Arithmetic:
    - +,-,*,/,%
    - ++,--,*=,...
  - Relational: <,>,<=,>=,==,!=
  - Logical: &&, ||, !, ? :
  - Bit: &,|,^,!,<<,>>

Common Syntax with Java

- Control structures:
  - if( ){ } else { }
  - while( ){ }
  - do { } while( )
  - for(i=0; i<100; i++){ }
  - switch( ) { case 0: ... }
  - break, continue, return

Differences from Java

- No exceptions
  - You must explicitly check for errors and propagate them
- No garbage collection
  - You must explicitly allocate and deallocate memory
- Pointers!
  - Directly manipulate the contents of memory

Primitive Types

- Integer types:
  - char : characters or one byte
  - int, short and long : integers of different size
  - can be signed or unsigned
- Floating point types: float and double
- No boolean type
  - 0 ⇒ false
  - ≠0 ⇒ true

Examples

- `char c='A';`
- `char c=100;`
- `int i=-2343234;`
- `unsigned int ui=100000000;`
- `float pi=3.14;`
- `double long_pi=0.31415e+1;`

Printing output

printf(format,param1, ...)

- format: string containing special markers where parameter values will be substituted
- %d for int
- %c for char
- %f for float
- %s for string

- Example:
  - printf(”Class %s: Size %d.\n", "CS4410", 999);
- **Warning:** mismatching markers and parameters can crash your program!

Enumerated Types

```
enum months{
  JANUARY,
  FEBRUARY,
  MARCH
};

enum months2{
  JANUARY=1,
  JULY=7,
  AUGUST
};
```

- Each element gets an incremented integer value, beginning with 0.
- Explicitly assigning a value affects following elements (AUGUST==8)

Memory Operations

- Pointers:

```
int a; /* An int */
int * ptr_a; /* A pointer to an int */
```

- The *value* of a pointer is the memory address it points to.
- Pointer operations:
  - '&' : obtain the address of a variable
  - '*' : *dereference* a memory address
- void* is a pointer to an unspecified type

Pointer example

```
int a;
int * ptr_a; /* ptr_a points to an
                undefined location */
ptr_a = &a;  /* ptr_a now points to
                integer a */
*ptr_a = 3;  /* variable pointed to
                by ptr_a is now 3 */
```

Memory Management

- **Global variables:**
  - Declared outside all functions.
  - Space allocated statically before execution
  - Space deallocated at program exit
  - Be careful about names across files:
    - Read up on **static** and **extern** variables

Memory Management

- **Local variables:**
  - Declared in the body of a function.
  - Space allocated on stack when entering the function (function call).
  - Initialization before function starts executing.
  - Space automatically deallocated when function returns, deleting the stack “frame”.
- **Warning:** referring to a local variable after the function has returned can crash your program!

```
int * bad_func(){
     int a = 37;
     return &a;
}
```

Memory Management

- **Heap variables:**
  - Memory is explicitly allocated via malloc() and deallocated via free()

```
void* malloc(int)
void free(void*)
```

  - Memory management is up to the program
- **Warning:** Calling free on a pointer more than once can crash your program!
  - Never calling free “leaks” memory.

Malloc/Free Example

```
int* ptr; /* pointer to an int */
/* allocate space to hold an int */
ptr = (int*) malloc(sizeof(int));

/* check if successful */
if (ptr == NULL) exit(1);
*ptr = 4; /* store value 4 */
printf("ptr: %p %d\n",ptr,*ptr);
free(ptr); /* deallocate memory */
```

Warning

- Dereferencing an un-initialized pointer can crash your program (or worse)!
- Consider initializing a pointer to NULL and checking before dereferencing.
- Some functions return NULL on error
  - Pay attention to the function specification!
  - Check return values!

Arrays and Strings

- Arrays:

```
/* declare and allocate space for array A */
int A[10];
for (int i=0; i<10; i++)
    A[i]=0;
```

- Strings: arrays of char terminated by \0

```
char[] name="CS4410";
name[5]='1';
```

- Functions to operate on strings in string.h
  - strcpy, strcmp, strcat, strstr, strchr.

Functions

- Arguments can be passed:
  - by value: a copy of the value of the parameter passed to the function
  - by reference: a pointer to the parameter variable is passed to the function
- Returned values from functions: by value or by reference.

Pass by Value/Reference

```
/* pass by value */
void swap(int n1, int n2){
   int temp;
   temp = n1;
   n1 = n2;
   n2 = temp;
}
/* pass by reference */
void swap(int* p1, int* p2){
   int temp;
   temp = *p1;
   *p1 = *p2;
   *p2 = temp;
}
```

- Modifying n1 and n2 only changes the local variables.
- To write a function that modifies its arguments, use references.

Function Pointers

```
void myproc(int d){
   ... /* do something */
}
void mycaller(void (*f)(int), int param){
   f(param); /* call function f with param */
}
void main(void){
   myproc(10); /* call myproc */
   mycaller(myproc, 10); /* call using mycaller */
}
```

Structures

```
struct birthday {
  char* name;
  enum months month;
  int day;
  int year;
};
struct birthday mybirthday =
  {"xyz",1,1,1990};
char initial = mybirthday.name[0];
mybirthday.month = FEBRUARY;
```

Structures

- Field types can be any type already defined.
- Example :

```
struct list_elem{
  int data;
  struct list_elem* next;
};
struct list_elem le={ 10, NULL };
struct list_elem* ptr_le = &le;
printf("The data is %d\n", ptr_le->data);
```

Typedef

- Creates an *alias* for a type
- Syntax: *typedef type alias;*
- Example:

```
typedef struct list_elem{
    int data;
    struct list_elem* next;
} list_elem;
list_elem le={ 10, NULL };
```

Preprocessor

- Headers

```
#include <stdio.h>
#include "myheader.h"
```

- Compile-time constants

```
#define MAX_LIST_LENGTH 100
```

- Conditional compilation

```
#ifdef DEBUG
printf("DEBUG: at line " __LINE__ ".\n");
#endif
```

Style

- Comment your code!
  - Especially when it's complex
- Don't bury arcane magic numbers in the body of your program
  - Create well-named constants
- Organize code logically
  - Pick a style and stick with it
  - Use descriptive function and variable names
  - Split large functions into manageable subroutines
  - Don't introduce unnecessary dependencies

Build Tools and Version Control

- Build systems
  - Organize compilation commands and dependencies
  - Enable incremental compiling
  - Examples: make, pmake, scons, etc
- Version Control
  - Keep track of changes
  - Simplifies project management among multiple developers
  - Examples: Subversion, Git, CVS, Mercurial

Summary

- C is great!
- Learn by doing
- Respect the power of C
  - Initialize variables before use
  - Don’t return pointers to local variables
  - Allocate and deallocate memory properly
  - Check return values

Don’t turn into this guy